# 15-411: First-Class Functions

Jan Hoffmann

# Function Pointers

# C1

## C1 is a conservative extension of C0

- (A limited form of) function pointers
- Break and continue statements
- Generic pointers (void*)
- More details in the C0 language specification

# Function Pointers in C

- In C we can use the address of operator & to get the address of a functions
- However, we cannot modify the content of a function's address
- Function types are defined using typedef

**Example:**

```
typedef int optype(int,int);

typedef int (*optype_pt)(int,int);
```

# Function Pointers in C

- In C we can use the address of operator & to get the address of a functions
- However, we cannot modify the content of a function's address
- Function types are defined using typedef

**Example:**

```
typedef int optype(int,int);

typedef int (*optype_pt)(int,int);
```

Not in C1!

# Function Pointers in C: Examples

```
int f (int x, int y) {
  return x+y;
}

int (*g)(int x, int y) = &f;

int main () {
  (*g)(1,2);
}
```

```
int f (int x, int y) {
  int g (int y) {return 0};
  return x+y;
}
```

# Function Pointers in C: Examples

```
int f (int x, int y) {
  return x+y;
}

int (*g)(int x, int y) = &f;

int main () {
  (*g)(1,2);
}
```

Not in C1!

```
int f (int x, int y) {
  int g (int y) {return 0};
  return x+y;
}
```

# Function Pointers in C: Examples

```
int f (int x, int y) {
  return x+y;
}

int (*g)(int x, int y) = &f;

int main () {
  (*g)(1,2);
}
```

Not in C1!

**Cannot define local functions:**

```
int f (int x, int y) {
  int g (int y) {return 0};
  return x+y;
}
```

# Function Pointers in C: Examples

```
typedef int optype(int,int);

int add (int x, int y) {return x+y;}

int mult (int x, int y) {return x*y;}

optype* f1 (int x) {
  optype* g;
  if (x)
    {g = &add;}
  else
    {g = &mult;}
  return g;
}

int g1 (optype* f, int x, int y) {
  return (*f)(x,y);
}
```

# Function Pointers in C: Examples

```
typedef int optype(int,int);

int h () {
  optype f2;
  int x = f2(1,2);
  return 0;
}
```

# Function Pointers in C: Examples

```
typedef int optype(int,int);

int h () {
  optype f2;
  int x = f2(1,2);
  return 0;
}
```

In C, variables can have a function type.

# Function Pointers in C: Examples

```
typedef int optype(int,int);

int h () {
  optype f2;
  int x = f2(1,2);
  return 0;
}
```

In C, variables can have a function type.

What happens if you compile the program?

# Function Pointers in C1

gdef ::= …
    | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

# Function Pointers in C1

gdef ::= …
    | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

unop ::= … | &

exp ::= … | (* exp) ( exp, … ,exp )

# Function Pointers in C1

gdef ::= …
    | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

unop ::= … | &

Can only be applied to functions.

exp ::= … | (* exp) ( exp, … ,exp )

# Function Pointers in C1

gdef ::= …
    | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

unop ::= … | &

Can only be applied to functions.

exp ::= … | (* exp) ( exp, … ,exp )

Dereference only in function application.

# Function Pointers in C1

gdef ::= …
    | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

unop ::= … | &

Can only be applied to functions.

exp ::= … | (* exp) ( exp, … ,exp )

Dereference only in function application.

**Small types:**

int, bool, t*, t[]

**Large types:**

struct s, ftp

# Function Pointers in C1

gdef ::= …
  | *typedef* type ftp (type vid, … , type vid)

type ::= … | ftp

unop ::= … | &

Can only be applied to functions.

exp ::= … | (* exp) ( exp, … ,exp )

Dereference only in function application.

**Small types:**

int, bool, t*, t[]

**Large types:**

struct s, ftp

No variables, arguments, and return values of large type.

# Static Semantics

$$\frac{ft = (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma(f) = ft}{\Gamma \vdash \&f : ft*}$$

$$\frac{ft = (\tau_1, \ldots, \tau_n) \to \tau \quad \Gamma \vdash e : ft* \quad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash *e(e_1, \ldots, e_n) : \tau}$$

# Dynamic Semantics

# Dynamic Semantics: Function Pointers

$$S; \eta \vdash (*e)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e \blacktriangleright ((*\_)(e_1, e_2)\ ,\ K)$$

$$S; \eta \vdash \&f \blacktriangleright (*\_)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e_1 \blacktriangleright (f(\_, e_2)\ ,\ K)$$

| | | | |
|---|---|---|---|
| Expressions | $e$ | $::=$ | $c \mid e_1 \odot e_2 \mid \mathsf{true} \mid \mathsf{false} \mid e_1 \ \&\& \ e_2 \mid x \mid f(e_1, e_2) \mid f(\,)$ |
| Statements | $s$ | $::=$ | $\mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{assign}(x, e) \mid \mathsf{decl}(x, \tau, s)$ |
| | | $\mid$ | $\mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \mid \mathsf{return}(e) \mid \mathsf{assert}(e)$ |
| Values | $v$ | $::=$ | $c \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nothing}$ |
| Environments | $\eta$ | $::=$ | $\cdot \mid \eta, x \mapsto c$ |
| Stacks | $S$ | $::=$ | $\cdot \mid S \ , \langle \eta, K \rangle$ |
| Cont. frames | $\phi$ | $::=$ | $\_ \odot e \mid c \odot \_ \mid \_ \ \&\& \ e \mid f(\_, e) \mid f(c, \_)$ |
| | | $\mid$ | $s \mid \mathsf{assign}(x, \_) \mid \mathsf{if}(\_, s_1, s_2) \mid \mathsf{return}(\_) \mid \mathsf{assert}(\_)$ |
| Continuations | $K$ | $::=$ | $\cdot \mid \phi \ , K$ |
| Exceptions | $E$ | $::=$ | $\mathsf{arith} \mid \mathsf{abort} \mid \mathsf{mem}$ |

# Summary I

All ops.

| | | | |
|---|---|---|---|
| Expressions | $e$ | $::=$ | $c \mid e_1 \odot e_2 \mid \mathsf{true} \mid \mathsf{false} \mid e_1 \;\&\&\; e_2 \mid x \mid f(e_1, e_2) \mid f()$ |
| Statements | $s$ | $::=$ | $\mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{assign}(x, e) \mid \mathsf{decl}(x, \tau, s)$ |
| | | $\mid$ | $\mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \mid \mathsf{return}(e) \mid \mathsf{assert}(e)$ |
| Values | $v$ | $::=$ | $c \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nothing}$ |
| Environments | $\eta$ | $::=$ | $\cdot \mid \eta, x \mapsto c$ |
| Stacks | $S$ | $::=$ | $\cdot \mid S\ ,\ \langle \eta, K \rangle$ |
| Cont. frames | $\phi$ | $::=$ | $\_ \odot e \mid c \odot \_ \mid \_ \;\&\&\; e \mid f(\_, e) \mid f(c, \_)$ |
| | | $\mid$ | $s \mid \mathsf{assign}(x, \_) \mid \mathsf{if}(\_, s_1, s_2) \mid \mathsf{return}(\_) \mid \mathsf{assert}(\_)$ |
| Continuations | $K$ | $::=$ | $\cdot \mid \phi\ ,\ K$ |
| Exceptions | $E$ | $::=$ | $\mathsf{arith} \mid \mathsf{abort} \mid \mathsf{mem}$ |

# Summary I

$$
\begin{array}{lll}
S\,;\eta \vdash e_1 \odot e_2 \rhd K & \longrightarrow & S\,;\eta \vdash e_1 \rhd (\_ \odot e_2\ ,\ K) \\
S\,;\eta \vdash c_1 \rhd (\_ \odot e_2\ ,\ K) & \longrightarrow & S\,;\eta \vdash e_2 \rhd (c_1 \odot \_\ ,\ K) \\
S\,;\eta \vdash c_2 \rhd (c_1 \odot \_\ ,\ K) & \longrightarrow & S\,;\eta \vdash c \rhd K \qquad (c = c_1 \odot c_2) \\
S\,;\eta \vdash c_2 \rhd (c_1 \odot \_\ ,\ K) & \longrightarrow & \mathsf{exception}(\mathsf{arith}) \qquad (c_1 \odot c_2 \text{ undefined}) \\
\\
S\,;\eta \vdash e_1\ \texttt{\&\&}\ e_2 \rhd K & \longrightarrow & S\,;\eta \vdash e_1 \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) \\
S\,;\eta \vdash \mathsf{false} \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) & \longrightarrow & S\,;\eta \vdash \mathsf{false} \rhd K \\
S\,;\eta \vdash \mathsf{true} \rhd (\_\ \texttt{\&\&}\ e_2\ ,\ K) & \longrightarrow & S\,;\eta \vdash e_2 \rhd K \\
\\
S\,;\eta \vdash x \rhd K & \longrightarrow & S\,;\eta \vdash \eta(x) \rhd K
\end{array}
$$

# Summary: Expressions

$$
\begin{array}{lcl}
S \; ; \eta \vdash \mathsf{nop} \blacktriangleright (s \; , K) & \longrightarrow & S \; ; \eta \vdash s \blacktriangleright K \\
S \; ; \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K & \longrightarrow & S \; ; \eta \vdash e \rhd (\mathsf{assign}(x, \_) \; , K) \\
S \; ; \eta \vdash c \rhd (\mathsf{assign}(x, \_) \; , K) & \longrightarrow & S \; ; \eta[x \mapsto c] \vdash \mathsf{nop} \blacktriangleright K \\
\\
S \; ; \eta \vdash \mathsf{decl}(x, \tau, s) \blacktriangleright K & \longrightarrow & S \; ; \eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K \\
\\
S \; ; \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & S \; ; \eta \vdash e \rhd (\mathsf{assert}(\_) \; , K) \\
S \; ; \eta \vdash \mathsf{true} \rhd (\mathsf{assert}(\_) \; , K) & \longrightarrow & S \; ; \eta \vdash \mathsf{nop} \blacktriangleright K \\
S \; ; \eta \vdash \mathsf{false} \rhd (\mathsf{assert}(\_) \; , K) & \longrightarrow & \mathsf{exception}(\mathsf{abort}) \\
\\
S \; ; \eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K & \longrightarrow & S \; ; \eta \vdash e \rhd (\mathsf{if}(\_, s_1, s_2) \; , K) \\
S \; ; \eta \vdash \mathsf{true} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & S \; ; \eta \vdash s_1 \blacktriangleright K \\
S \; ; \eta \vdash \mathsf{false} \rhd (\mathsf{if}(\_, s_1, s_2), K) & \longrightarrow & S \; ; \eta \vdash s_2 \blacktriangleright K \\
\\
S \; ; \eta \vdash \mathsf{while}(e, s) \blacktriangleright K & \longrightarrow & S \; ; \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K
\end{array}
$$

# Summary: Statements

$$S \; ; \eta \vdash f(e_1, e_2) \rhd K \quad\longrightarrow\quad S \; ; \eta \vdash e_1 \rhd (f(\_, e_2) \; , K)$$
$$S \; ; \eta \vdash c_1 \rhd (f(\_, e_2) \; , K) \quad\longrightarrow\quad S \; ; \eta \vdash e_2 \rhd (f(c_1, \_) \; , K)$$
$$S \; ; \eta \vdash c_2 \rhd (f(c_1, \_) \; , K) \quad\longrightarrow\quad (S \; , \langle \eta, K \rangle) \; ; [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot$$

*(given that $f$ is defined as $f(x_1, x_2)\{s\}$)*

$$S \; ; \eta \vdash f(\,) \rhd K \quad\longrightarrow\quad (S \; , \langle \eta, K \rangle) \; ; \cdot \vdash s \blacktriangleright \cdot$$

*(given that $f$ is defined as $f(\,)\{s\}$)*

$$S \; ; \eta \vdash \mathsf{return}(e) \blacktriangleright K \quad\longrightarrow\quad S \; ; \eta \vdash e \rhd (\mathsf{return}(\_) \; , K)$$
$$(S \; , \langle \eta', K' \rangle) \; ; \eta \vdash v \rhd (\mathsf{return}(\_) \; , K) \quad\longrightarrow\quad S \; ; \eta' \vdash v \rhd K'$$
$$\cdot \; ; \eta \vdash c \rhd (\mathsf{return}(\_) \; , K) \quad\longrightarrow\quad \mathsf{value}(c)$$

# Summary: Functions

# Dynamic Semantics: Function Pointers

$$S; \eta \vdash (*e)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e \blacktriangleright ((*\_)(e_1, e_2)\ ,\ K)$$

$$S; \eta \vdash \texttt{\&}f \blacktriangleright (*\_)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e_1 \blacktriangleright (f(\_, e_2)\ ,\ K)$$

# Nominal Types

## C1 treats function types nominally

```
typedef int optype1(int,int);

typedef int optype2(int,int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```
int add (int x, int y) {return x+y;}

int main {
  optype1* f = &add;
  optype2* f = &add;
  return 0;
}
```

# Nominal Types

## C1 treats function types nominally

```
typedef int optype1(int,int);

typedef int optype2(int,int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```
int add (int x, int y) {return x+y;}

int main {
  optype1* f = &add;
  optype2* f = &add;
  return 0;
}
```

Like null, add can have both types.

# Nominal Types

## C1 treats function types nominally

```
typedef int optype1(int,int);

typedef int optype2(int,int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```
int add (int x, int y) {return x+y;}

int main {
  optype1* f = &add;
  optype2* f = &add;
  return 0;
}
```

Like null, add can have both types.

```
(*&add)(x,y)
```

# Nominal Types

**C1 treats function types nominally**

```
typedef int optype1(int,int);

typedef int optype2(int,int);
```

optype1 and optype2 are different types and pointers of optype1 and optype2 cannot be compared.

```
int add (int x, int y) {return x+y;}

int main {
  optype1* f = &add;
  optype2* f = &add;
  return 0;
}
```

Like null, add can have both types.

```
(*&add)(x,y)
```

Not allowed in C1.

# Nominal Type and Contracts

```
typedef int binop_fn(int x, int y);
  //@requires x >= y; ensures \result > 0;
typedef int binop_fn_2(int x, int y);
  //@requires x != y;
```

- binop_fn and binop_fn_2 are treated as different types
- The  call *f(3,3) can cause a precondition violation
- The call *f2(3,3) might be fine even if f and f2 point to the same function

# First-Class Functions

# Currying and Partial Application

In ML we can have functions that return functions

```
let f = fn (x, y) => x + y
let g = fn x => fn y => f (x, y)
let h = g 7
```

# Currying and Partial Application

In ML we can have functions that return functions

```
let f = fn (x, y) => x + y
let g = fn x => fn y => f (x, y)
let h = g 7
```

In C (C0, C1, …) we could support this by adding a new syntactic form for anonymous functions

```
fn (int i) { stm }
```

# Example

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

# Dynamic Semantics of Anonymous Functions

## Dynamic semantics is not immediately clear

In a functional language we could define the semantics using substitution

`addn(7)` would lead to

```
return fn (int y) { return 7 + 8 + y; }
```

# Dynamic Semantics of Anonymous Functions

## Dynamic semantics is not immediately clear

In a functional language we could define the semantics using substitution

`addn(7)` would lead to

```
return fn (int y) { return 7 + 8 + y; }
```

But in an imperative language that does not work

The variable x might be incremented inside a loop
What would the effect of the substitution be?

# C1 Example: Dynamic Semantics

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

# C1 Example: Dynamic Semantics

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

When we call addn the values of x and z are available.

# C1 Example: Dynamic Semantics

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
```

When we call addn the values of x and z are available.

Of course, function arguments are not available statically.

# C1 Example: Dynamic Semantics

```
unop_fn* addn(int x) {
   int z = x + 1;
   return fn (int y) { return x + z + y; };
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
```

When we call addn the values of x and z are available.

Of course, function arguments are not available statically.

**Idea: Store variable environment with function code**

**➡ function closure**

# Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

$$S; \eta \vdash \texttt{fn}(x, y)\{s\} \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta \rangle\!\rangle \blacktriangleright K$$

$$S; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle \blacktriangleright (*\_)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2) \ , K)$$

$$S; \eta \vdash v_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2) \ , K) \quad \longrightarrow \quad S; \eta \vdash e_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_) \ , K)$$

$$S; \eta \vdash v_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_) \ , K) \quad \longrightarrow \quad S; \langle \eta, K \rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \triangleright \cdot$$

# Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

New value: function closure.

$$S; \eta \vdash \texttt{fn}(x, y)\{s\} \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta \rangle\!\rangle \blacktriangleright K$$

$$S; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle \blacktriangleright (*\_)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S; \eta \vdash e_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2) \ , K)$$

$$S; \eta \vdash v_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2) \ , K) \quad \longrightarrow \quad S; \eta \vdash e_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_) \ , K)$$

$$S; \eta \vdash v_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_) \ , K) \quad \longrightarrow \quad S; \langle \eta, K \rangle; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \triangleright \cdot$$

# Function Closures: Dynamic Semantics

For functions with two arguments (other functions are similar)

New value: function closure.

Store the current variable environment.

$$S ; \eta \vdash \texttt{fn}(x, y)\{s\} \blacktriangleright K \quad \longrightarrow \quad S ; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta \rangle\!\rangle \blacktriangleright K$$

$$S ; \eta \vdash \langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle \blacktriangleright (*\_)(e_1, e_2) \blacktriangleright K \quad \longrightarrow \quad S ; \eta \vdash e_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2)\ ,\ K)$$

$$S ; \eta \vdash v_1 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(\_, e_2)\ ,\ K) \quad \longrightarrow \quad S ; \eta \vdash e_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_)\ ,\ K)$$

$$S ; \eta \vdash v_2 \blacktriangleright ((*\langle\!\langle \texttt{fn}(x, y)\{s\}, \eta' \rangle\!\rangle)(v_1, \_)\ ,\ K) \quad \longrightarrow \quad S ; \langle \eta, K \rangle ; [\eta', x \mapsto v_1, y \mapsto v_2] \vdash s \triangleright \cdot$$

# Another Example

```
unop_fn* addn(int x) {
   unop_fn* f = fn (int y) { x++; return x + y; };
   x++;
   return f;
}

int main() {
   unop_fn* h1 = addn(7);
   unop_fn* h2 = addn(6);
   return (*h1)(3) + (*h1)(5) + (*h2)(3);
}
```

# Function Closures in Python

```
def makeInc(x):
  def inc(y):
     # x = x + 1
     return y + x
  x = x + 1
  return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
```

# Function Closures in Python

```
def makeInc(x):
  def inc(y):
     # x = x + 1
     return y + x
  x = x + 1
  return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
```

What's the return value?

# Function Closures in Python

```
def makeInc(x):
  def inc(y):
     # x = x + 1
     return y + x
  x = x + 1
  return inc

inc5 = makeInc(5)
inc10 = makeInc(10)

inc5(4)
```

What happens when we add this line?

What's the return value?

# Implementing Functions Closures

- Need to store variable environment and function body
- Difficulty: We cannot determine statically what the shape of the environment is
- Similar to adding a struct to the function body
- Store all variables that are captured by the function closure on the heap